WINDOWS EDITION

Python

Novice to Ninja

bruce smith

Learn Python the Windows Way

WINDOWS EDITION

PYTHON Novice to Ninja

ISBN: 9781764000109

Thanks to: Joshua Jones, Tom Racy Annie James

EPILEPSY

I live with epilepsy. It's something most people know a little about, but often not the whole story. For many, the word brings to mind flashing lights and dramatic seizures — but that's only one small part of a much bigger picture. In truth, there are many different types of epilepsy, and they affect each person in very individual ways. Some are invisible to others; some are unpredictable; all deserve understanding.

I'm sharing this not for sympathy, but for awareness. Epilepsy can shape a life, yet it doesn't have to define it. With support, awareness, and a bit of patience, people with epilepsy can lead full, creative, and active lives.

If speaking about it helps even one person feel less alone, or helps someone else better understand a friend, colleague, or family member with epilepsy — then it's worth every word.

The one good is that having epilepsy proves I've got brain whatever anyone may say.

www.codingpress.com.au

www.brucesmith.info

CONTENTS

Preface

Many people assume that Python is identical no matter which machine it runs on. In truth, the core language is the same everywhere — if you write a loop, a function, or an import statement, it will behave the same on Windows, macOS, or Linux. That's by design. Python is meant to be portable and easy to share across platforms.

However, what often trips people up are the operating system specifics. A few common examples:

File paths: Windows uses backslashes (C:\Users\Bruce\Documents) while Linux and macOS use forward slashes (/home/bruce/Documents). A simple hard-coded path can cause a program to fail.

Environment variables and commands: Activating a virtual environment, setting a PATH variable, or running Python from the terminal is slightly different on Windows compared to macOS or Linux.

Installed tools and libraries: Some packages with compiled code (like NumPy or SciPy) may behave differently depending on the underlying OS, especially if you're using Windows on ARM hardware.

Default shells: Windows offers PowerShell and CMD, while Unix-like systems rely on bash or zsh. The same command may need different wording in each shell.

So, while your Python programs may be "portable," the way you set them up and run them is not always identical. This book is written with that in mind. It's about making Python work smoothly on a Windows PCs, taking into account the quirks of PowerShell,, the Windows filesystem, and the way Windows handles installation and configuration. My aim is to make sure you don't waste time on platform issues and can focus on learning Python itself.

IMPORTANT NOTE

Python programs follow a style guideline known as PEP8, which will be covered later in this book. Although PEP8 helps maintain consistency, it is not required for a program to run correctly. One of the guidelines suggests including two blank lines between certain sections of code, but this will not always be followed in the listings here. This is to reduce unnecessary whitespace and improve readability. Editors typically colour Python code, which makes it visually clear on screen. In some cases, the code in this book may not fully comply with PEP8 for the sake of clarity and presentation (essentially space on the page).

PROGRAMS AND QUESTIONS

There are over 250 programs in this book. I suggest that you type in as many of these as you can. It is the best way to learn and get a feeling for the commands, syntax and flow of a Python program. Learn how to correct and ensure your coding is correct and works. Bear in mind that some program lines extend longer than the space available across the width of the book. As such they will often 'wrap-around' onto the next line. Be aware of this when you write your code. But another good way to get to terms with what is on each program line! Note also that, where possible, the comment lines are listed in the book in italics.

That said the programs are available to download from the website:

```
www.brucesmith.info
```

Additional information may also be here in relation to the book and Python updates.

QUESTIONS AND EXERCISES

Most chapters, not all, contain questions at the end. These questions are related to the text in the chapter just read. There are up to 15 questions. No answers are provided in the book, they are there for you to answer or review the contents of the chapter prior to answering.

00: Novice to Ninja

What does **novice** and **ninja** mean regarding learning to program Python on Windows? There are many books for programmers who are starting their Python experience. But they cover the basics and **don't** get under the 'hood' and into the detail. They leave you in the hallway, and don't show the rest of the house. Within these pages I seek to correct that and take you to those places beyond the hallway, providing a higher level of knowledge and expertise. Transforming your from a complete beginner (or intermediate), who's just learning the ropes, to a skilled and confident Python coder. Merriam-Webster defines each word thus:

- A **novice** in computer terms can be defined as a beginner or someone who has no previous experience in a particular field or activity.
- On the other hand, **ninja** isn't typically defined in the same way in dictionaries. In popular usage, it's often used to describe someone who has achieved a high level of skill or expertise. An expert, maybe someone to be feared?

You may not be a novice in the use of Windows, and may have some programming knowledge, but this tome will just accelerate your learning. Being a '**Python ninja**' doesn't mean you've mastered every single aspect of the language, after all, there's always more to learn. Instead, it means you've reached a point where you're comfortable and efficient with the language, can solve problems creatively, and write clean, effective code. You know how to handle different challenges, think like a programmer, and confidently create your own projects and solutions or collaborate with others.

Programs

There are a **lot** of programs in this book. You can download the source from the authors website at:

```
www.brucesmith.info
```

They are there for your convenience. I would **strongly** suggest that you **type these programs in yourself.** Unless you do you won't start to understand how a Python program goes together. How it is structured and what goes where as well as why? Most of demo programs are not that long, so it shouldn't be overly difficult. As you progress through the book then they will become much longer, so you could delve into the download at that point.

Typing is another key skill. If you can't type, to whatever degree, in today's world then things become long-winded. It is the biggest skill you can master and is starting to become part of the school curriculum here.

```
feedback@brucesmith.info
```

Use the Website Please

If you didn't buy this through **The Coding Press** website, I'd be grateful if you consider purchasing any more of my books from there. You will find a larger choice of book formats and help support me directly as a creative.

This means I can fully reap the benefits of my efforts in writing and publishing. Supporting creators directly allows us to continue producing more great content. People often think of the cost of the final project but not the six plus months or so it takes to go from first word to finished article!

01: Hello

Python, alongside JavaScript and Java, is one of the most widely used programming languages in the world. Some might even say it's the most popular and significant, especially in the business world where it's the go-to software. If you're aiming to become a commercial programmer and haven't yet explored Python, you might find fewer doors open—**it's that essential**.

As a taster, look at these famous organisations and their uses of Python:

- **YouTube:** The world's largest video-sharing platform, extensively uses Python for back-end services, including video sharing, website operation, and system administration. Python is known for being simple and easy to maintain, making it ideal for a platform like YouTube, which requires handling massive amounts of data and user interactions efficiently. Its strong libraries for web development, and support for data handling allow YouTube engineers to scale the platform easily.
- **Instagram**: one of the most popular social media platforms, relies heavily on Python and its modules, for handling millions of active users and managing its back-end services. Instagram chose Python for its simplicity and ability to help developers write clean, maintainable code. It also helps Instagram scale its infrastructure efficiently. Python's scalability and speed in development cycles allowed Instagram to keep up with its explosive growth without compromising performance.
- **Spotify:** the popular music streaming service uses Python for data analysis, back-end services, and machine learning to provide personalised recommendations. Python excels at handling large amounts of data, which Spotify needs for features like personalised music recommendations and user behaviour analysis. Its data science libraries, such as Pandas and Scikit-learn, make it ideal for analytic

and machine learning tasks. Additionally, Python's asynchronous framework capabilities, like Tornado and asyncio, enable Spotify to handle multiple concurrent connections (such as streaming requests) efficiently.

- **Reddit:** is one of the largest online communities, is primarily written in Python. It uses Python for its back-end to manage user submissions, interactions, and content. Reddit originally started with Lisp but migrated to Python for its simplicity and wide range of libraries. Python allows Reddit to scale easily, handle millions of daily interactions, and manage a large amount of content without sacrificing performance. Python's versatility and Reddit's use of frameworks enable it to support its massive user base while remaining flexible for future growth. Reddit's decision to use Python also makes it easier to maintain and add new features over time.

- **Google:** has used Python since its early days, and it plays a significant role in various parts of Google's infrastructure, including search algorithms, system management tools, and back-end services. Google values Python for its simplicity, speed of development, and readability. These attributes allow developers to write and maintain code quicker, which is crucial in a large-scale environment like Google. Python's flexibility also allows it to be used in everything from system administration to machine learning. For example, Google's internal systems (like parts of Google Search) and tools like YouTube Data API rely on Python. In fact, Guido van Rossum, the creator of Python, worked at Google for several years, and Google actively supports Python's development.

- **Netflix:** utilises Python for content delivery, data analytic, and automation, playing a critical role in its recommendation algorithms and internal systems. Netflix uses Python for data streaming and analysis to track user preferences and optimise content recommendations. Python's powerful libraries, such as NumPy, Pandas, and TensorFlow. Additionally, Python helps Netflix automate content delivery and infrastructure management, making their systems more efficient and scalable.

- **NASA** uses Python in various scientific computing and space research applications, including data analysis and simulations. Python's extensive scientific libraries (like SciPy and NumPy) and its ease of integration with other technologies make it ideal for complex scientific tasks. Python is widely used in scientific research because of its readability and vast array of scientific libraries. Python's ease of integration with other languages (such as C or Fortran for performance-critical code) also makes it an ideal choice for NASA,

where various specialised tools and systems need to work together seamlessly.

- **Uber:** uses Python for back-end services and data science tasks, helping manage its large-scale ride-sharing operations. Python's ease of use and ability to handle large-scale, real-time data processing makes it a natural fit for Uber's, fast-paced environment. Uber processes millions of ride requests, driver updates, and trip calculations in real-time, and Python's capabilities help manage these concurrent processes efficiently. Python also plays a critical role in Uber's data science efforts, where it's used for calculating estimated time of arrival (ETA), optimising routes, and pricing algorithms.

Windows Versions

The contents of this book have been tested on various versions of Windows, including Windows 7, 8, 10, and 11, across both Home and Pro editions, as well as Windows running on ARM-based devices. Python 3 behaves consistently across these versions, but there are some subtle differences worth noting—especially when it comes to performance, installation quirks, or library compatibility. Where these differences matter, I've made sure to point them out with any relevant guidance.

In general, if your version of Windows can install and run Python 3, then this book will be applicable. Below is a guide to how Python 3 performs on various Windows versions.

Windows 7 (2009): Python 3 can still be installed on Windows 7, though official support from Python.org ended with Python 3.8. You may encounter issues with installing newer packages or using modern tools, especially pip or virtual environments. It's usable but not recommended for beginners starting fresh.

Windows 8 / 8.1 (2012–2013): Python 3.9 and earlier will run fine on Windows 8.1, but Windows 8 (not 8.1) has patchy support. Performance is decent, and Python scripts and GUI programs will run, but compatibility with recent Python modules may be limited.

Windows 10 (2015): This is where Python 3 truly thrives. Windows 10 supports all current Python versions (up to 3.12 at time of writing), and the Microsoft Store even offers a simple installer. Performance is solid across most hardware, and all libraries and tools are fully supported.

Windows 10 on ARM (Surface devices etc.):
Python 3 runs here using a special ARM64 version. While most pure Python scripts will work fine, some C-based packages may not install or will need

alternative builds. Python performance on ARM is improving, but still slightly behind mainstream desktop versions.

Windows 11 (2021):
Python 3 runs beautifully on Windows 11. This version offers full compatibility with the latest tools, packages, and libraries.

Windows 11 on ARM (e.g. Surface Pro X):
Python 3 can run via the native ARM64 builds or x64 emulation. Pure Python code runs well, but again, some performance-sensitive libraries may need ARM-compatible builds. Not ideal for heavy-duty data science yet, but great for learning and basic projects.

Windows Server Editions (Various):
Python 3 runs fine on Windows Server (2012 R2 and later). These systems are often used for automation scripts, web servers, and enterprise tools. Be aware that security policies and firewalls might need tweaking for package installation and updates.

Python Types

Although we just say "Python", there are a few different flavours of it. You don't need to know about all of them to use this book, but it helps to be aware that Python comes in different forms depending on where and how you're running it.

Python / CPython
This is the version of Python we're using—downloaded from python.org or the Windows Store. It's the standard one, written in C, and it's what most people mean when they say "Python".

Jython
A version of Python that works inside the Java world. Handy if you need to mix Python with Java programs, but not something we'll use here.

IronPython
This one's for the .NET world. It lets you write Python that can interact with C# and other Microsoft tools. Useful for advanced Windows users.

PyPy
A version of Python that runs faster—much faster in some cases. It's aimed at people who want better performance without changing their code. We'll talk about this in Chapter 44.

Stackless Python
Based on CPython,

Py Space

No wonder, then, that Python has found a cosy spot on most computers.

So, what makes Python so special? It's known as a high-level language, which means it's designed with us humans in mind—**easy to read and write**. High-level languages are user-friendly and more abstract compared to low-level languages. Programmers love them because the code is easy to understand and maintain. Fun fact: another high-level language, C, was used to create Python itself, which is why Python's official name is C Python.

And no, the name has nothing to do with snakes. Python is named after a cult 70s British comedy show, *Monty Python's Flying Circus*. Remember John Cleese's 'Ministry of Silly Walks'? Classic! (If you haven't seen it, give it a search on YouTube.) There are more nods to the show sprinkled throughout the language.

At first glance, Python code might look a bit intimidating. Don't let that fool you. Despite its appearance, it's all about **readability**. Python emphasises 'structured programming,' nudging you to write clean and tidy code. It's like the language itself is helping you craft perfect programs. Structure is key in every aspect of life, so why should a programming language be any different?

One of Python's biggest charms is the availability of ready-made building blocks for writing programs, kind of like assembling a house from bricks. Think of these bricks as the building blocks of the language. In Python,

they're called **libraries** and **modules**. They do exactly what they say on the tin—libraries of code with specific functions, and modules that provide exactly what you need.

The best part? You don't need to create each step from scratch. You simply pick the blocks you need and snap them together. Python's popularity ensures there are plenty of these resources available, all designed to make your coding life easier.

Python's syntax is simple and readable, making it a breeze for beginners. The way Python uses **line indentation** to define code blocks makes it easy to spot different parts of a program at a glance. Plus, Python allows you to execute commands and segments of code 'on the fly' allowing you see results immediately when typing commands at the prompt.

And let's not forget about the massive **'standard library'** that comes with Python—a treasure trove of pre-written code. You don't have to worry much about getting access to these tools because the standardised interface between them makes it super easy. The standard library itself is ever **expanding**. These modules such as NumPy that have normally had to be downloaded and installed, are now part of this library.

This not only saves you time but also keeps your programs lightweight by only using what you need. Python's **cross-platform compatibility** means your programs can run smoothly on various operating systems like Windows, macOS, and Linux. Write it once, use it many times.(however, see note on Compatibility in the Preface.)

So, as we move forward, the code we write on a Windows computer will be, for the most part, transferable to other environments. How great is that?

Python Uses

On the first page of this chapter, I outlined just some of the large multi-nationals who make use of Python everyday, indeed you could say it underpins a large chunk of their business functionality.

Python plays a pivotal role in the educational landscape of Windows, offering versatility beyond just coding for learning. Some of the remarkable applications of Python include:

Home Automation: Python is a go-to for automating and controlling smart home devices, enabling users to script interactions with sensors, cameras, lights, and more.

Making: With an extensive array of add-on like hats, robots, displays, and weather monitors, Python's libraries control these attachments seamlessly.

Web Development: Python serves as a capable tool for crafting web applications, making it ideal for web-based projects.

Game Development: Crafting simple games using Python and libraries like PyGame provides an enjoyable introduction to programming and game development.

IoT (Internet of Things): Python finds its niche in IoT projects, connecting sensors, actuators, and other IoT devices, facilitating communication with cloud services.

Data Science and Analytics: Many of the Python libraries support data science, machine learning, and analytic, empowering users to analyse data and run machine learning models.

Robotics: Python's prowess extends to programming robots and robotic systems and generally simplifies hardware control.

Network Programming: Python's networking capabilities make it apt for projects involving device communication over a network, such as building a networked media centre or a file server.

Security and Penetration Testing: Python is an asset for security-related tasks, offering tools and libraries for penetration testing and network security.

Python is not just a programming language; it's a gateway to a multitude of exciting possibilities across various domains. The applications are endless.

Home Help

I personally use Python for a lot of things at home. It's so easy to use. Anything that involves, sorting, figures, text etc. For example, I use it for pulling together all my account data for the end of year tax return.

For this book, I typeset it using an application called *Affinity Publisher*. I created a program that extracts all the programs in the text and then test each one of them. Any errors are logged and changes can be made. This would otherwise be a time consuming copy and paste process.Even the index was completed with Python.

It's up to you to come up with the ideas…

Learning

So, how do you go about learning Python? Well if you have this book you're well on your way. Given that fact there are some things to help:

Set Clear Goals

Decide why you want to learn Python: Do you want to use it for web development, data analysis, automation, or game development? Or do you

just want to **learn**? Knowing your end goal will help guide your learning path.

Small Steps. For example, aim to write a small Python script within the first week, then move to more complex projects over time.

Get the Basics Right

Understand fundamental programming concepts. It's important, especially for Python. Keep and open mind and ensure you understand one chapter before jumping to the next one. Answer any questions and try a programming example. Get though correct. Then move on. Take existing programs and rework them for something you need.

Modify existing code: Try tweaking open-source code to see how things work. It helps you understand how small changes affect the program.

Practice, Practice, Practice

Consistent coding is the key. Even 20–30 minutes a day will build your skills faster than cramming once a week.

Stay Curious

Be open to learning new things as Python is vast and versatile. Try solving problems. Get online and check out the Python communities and forums. Keep in touch with the Python community.

02: Python Interactive

Most Windows PCs don't come with Python installed, you'll have to install it. This is no different to installing a new application.

So how to do this? We'll need to get access to the bones of the computer. You're using Windows and this is itself a program that sits at the top of a hierarchy of software. Windows is based around a **GUI**, which is short for **Graphical User Interface**. It is a user interface that uses visual elements like windows, icons, buttons, and menus to enable interaction with a computer, rather than relying on typed commands. These items or widgets are all connected to the underlying software that runs the GUI. These essentially connect to the parts that make the computer run and carry out the actions you require.

First up, let's see what you have Python already installed installed already. Click the magnifying glass (search) icon in the ribbon bar at the bottom of the screen. Into that small window type:

```
Powershell
```

Figure 2a below , shows the ribbon bar at the base of the screen.

Figure 2a. Entering the Powershell command in the 'Search window..

A pop-up screen will appear and at the top of this will be an input bar. Ignore this for now. Once you have typed Powershell hit the **Enter** key.

A back window will appear. In here you'll see some text along the lines

```
Windows PowerShell
```

```
Copyright (C) Microsoft Corporation. All rights
reserved.
PS C:\Users\bruce>
```

The black window may also include the line:

```
Install the latest PowerShell for new features and
improvements! https://aka.ms/PSWindows
```

This just means you aren't using the latest version of Powershell. Ignore this for the time being.

The prompt: *`PS C:\Users\bruce>`*

shows your current folder. Any commands you run will happen in this location

Oh Come On! What About Python

Okay, you're impatient, I know, but we needed to get to this point and start At the command line (note that the case of letters is important so we use lowercase throughout) Type:

```
python - -version
```

Note: It's two hyphens with no space between them, but there is a space between 'python' and the first '-'. If Python is not installed the output will be alog the lines of

```
Python was not found; run without arguments to install
from the Microsoft Store, or disable this shortcut from
Settings
```

In other words, Python is **not** installed. However, if you just get a few numbers, then you have Python installed, and the numbers represent the version you have. For instance:

```
13.11.1
```

If you get a message other that then you don't have Python installed.

Installing Python

(IMPORTANT: Install Python from the Python website and **not** the Windows Play Store. The former provides additional features that the Play Store versio does not have at the time of writing.)

Here's how you install Python: Open a web browser and in the URL bar type:

```
python.org
```


This takes you to the official Python website. You can find out quite a lot here, so it is worth browsing at some point. But select the Downloads menu.

Here you'll see a 'Download Python' button with a number which represents the version of Python you will load down. This is the latest release. You might see below a table suggesting the date of the next release as well.

Click on the Download Python button. Python will now download onto your computer. Go to the download widget next to the photo of you in the menu bar and the dropdown here should show some simple to this:

Figure 2b : Downloading Python

Once installed, you'll get a window popping up as above (Figure 2b).

Note that at the bottom of the window there are two check boxes. Check them both. Then click 'Install Now'. Click Yes on to anything the installation may ask you.

Reopen the Powershell window once again and type:

```
python --version
```

at the prompt. Now you should see Python and the version number you just installed, It might be:

```
python 3.13.1
```

Or higher depending on the time of installation. Later versions, such as 1.15.1, were released after the book's publication.

At the command prompt type:

```
python
```

Now you'll get a couple of lines of text followed by a blue:

```
>>>
```

The **>>>** represents the Python prompt. So, anything you type here is expected to be a Python command. For this reason, it is called the Python Interactive Shell (we won't find an acronym for this one.) Python will action anything. So let's try something. Type:

```
print('Gday Mate!')
```

And you'll get the 'Gday Mate' echoed back to you. That's your first Python command! More on this in due course. Exit Python by typing

```
exit()
```

What happens if Python is already installed? If the version number is the same as the latest version (see Python.org) then you're okay. But if there is a newer version, you'll want to upgrade Python to the latest version. (That said, it's not imperative as long as the version number starts with a three.)

Download the Latest Python Version:

Here's how you can download the

- Visit the official Python website.
- Download the latest version (e.g., Python 3.x.x).
- Run the Installer:
- Open the downloaded installer.
- Select Upgrade Now if upgrading an existing installation.
- Ensure Add Python to PATH is checked during installation.
- Verify the Installation.

```
python --version
```

IDE

An **Integrated Development Environment**—what we in programming speak call an IDE—is pure bliss, like the best thing since **apple pie** and custard. So, let's get set up for some coding comfort. I recommend creating a folder on your Desktop. Maybe give it a name like 'PYTHON'. The name doesn't matter, but something specific will help keep your coding projects organised. If you've downloaded the program files from my website, pop them into this folder.

Python programs are simply text files, a bunch of statements that, when combined just right, perform tasks. We group these statements into sets, each set handling a specific job. Bundle them together, and bingo—you've got a program. Python programs have the postfix **py**.

The IDE is the hero of our coding adventure. Think of it as a magical space where you can create, run, and fix your programs, all within one window.

For this journey, our IDE of choice is 'Thonny', which is just perfect for beginners. You'll need to install in, but you're getting pretty good at this.

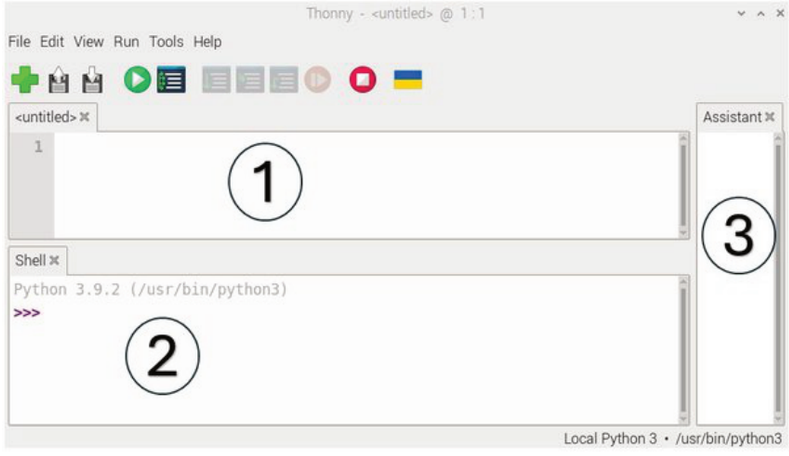
Go to the website at:

```
Thonny.org
```

and select the Download for Windows button, this will almost certainly be the one first in the list provided. It will say "recommended for you." Then follow the prompts. (Don't worry if your screen looks a bit different. Looks and layout can vary slightly from version to version. I've put the numbers there for descriptive purposes.)

Figure 2c Installing Thonny.

Maximise the Thonny window to fill your screen if you like, or just keep it as a normal floating window that you can resize as needed. The choice is yours. Thonny's interface is all about simplicity, designed to keep things smooth and easy as you write and run your Python code. While it might not be the top pick for larger projects, it works perfectly for tutorial purposes.



2d Typical Thonny start-up screen.

Let's break down the Thonny window (Figure 2c)into three mini windows, each with its own job. I've numbered them 1, 2, and 3 for easy reference:

1. **Script Editor**: This is where the magic happens. Write your Python code here. It's clean, it's spacious, and it's your coding canvas.

2. **Shell**: Just below the Script Editor is the Interactive Python Shell. This is your playground for testing snippets of code on the fly, like what we did in the command line earlier. We call this a **REPL** (Read-Eval-Print Loop).

3. **Assistant**: On the right, this window gives you feedback about your program when it runs. If there's an error, it'll offer some hints on what might need fixing.

You can resize any of these windows individually by dragging their borders.

Across the very top of the window, you'll find the menus as drop-downs and icons. These are laid out in a standard format, so they should feel familiar. Most are intuitive, but a few might make you pause—no worries, we'll explore their purposes as we go along. If you hover your pointer over the icons, a tool tip will pop up explaining what they do.

Figure 2e. The Thonny Menu Bar.

Now, if you glance at the bottom right-hand side of the Thonny window, you'll see the location where Python is installed. This can vary but it might look something like this:

```
Python 3.10.11
(C:\Users\bruce\AppData\Local\Programs\Thonny\python.
exe)
```

This indicates you which version/release of Python you are using and that you are using the version of Python that is integrated as part of Thonny.

If it is rads something else than that is the location of the Python edition you are using reads where it is located.

If you installed Python from the Microsoft Store, Windows places a shortcut here that points to the store version of Python. Even if you didn't install it via the Microsoft Store, Windows sometimes includes this alias, which redirects users to download Python when it's not installed.

Now, you're all set. The interface of Thonny. is a model of clarity, uncluttered and honed to the essential elements for Python coding. Thonny's user-friendly design boasts simplicity to ensure smooth writing and running of your programs. While it might not be the go-to IDE for larger, more complex programs, those are likely beyond the scope of this book.

Trying It Out

In Thonny, you can either use the existing program window ('1' above) or create a new one by clicking the green '+' icon. You can also go to **File** in the menu bar and select **New** or simply press '<Ctrl+N>' (meaning press the 'Ctrl' and 'N' keys together). There are so many ways to start—just pick one and get going! This action will open a new tab in the code area, and you can switch between tabs by clicking the one you need.

Type the following Python code in the editor:

```
print("Hello from Thonny!")
```

Save the file by clicking on 'File' and selecting 'Save', or by pressing <Ctrl+S>. Choose a filename and location for your Python program, making sure to give it a '.py' extension. For example, you could save it as:

```
hello.py
```

You'll see the name and directory of the file in the Thonny bar at the very top of the window.

To run the program, click on the green **Run** button in the toolbar, select 'Run' from the menu bar, or simply press the 'F5' key as a shortcut. I've put a

small yellow sticker on my F5 key—it makes it easy to find . Well worth doing. But any method is fine to run a program.

Thonny will execute the Python program, and you should see the output "Hello from Thonny!" displayed in the Shell pane at the bottom of the Thonny window.

If you look at the program window, you'll notice that the small tab might say **<untitled>*** at the top. This indicates that the file hasn't been saved yet. If it had the files name would be displayed here.

In case you're curious, Thonny draws its name from a fictional snake character known as '**Thonny the Python**'—kinda like the character on the front cover!

When writing Python programs, you can use either uppercase or lowercase characters, but there are some general conventions for when to use each. As we delve deeper into Python, we'll uncover and define these conventions.

A Real Program

Open a new code editor window by pressing the big green cross. Click in the Code Editor window and carefully enter the following:

```
#Filename: start1.py
import datetime
now = datetime.datetime.now()
print(now)
```

If you've downloaded the programs then this isn't included. You need to type it in. Now, let's bring this Python program to life. Hit the green '**Run**' button at the top of the window, and watch the result unfold in the '**Shell**' window. You should see the current date and time displayed down to six decimal points of a second!

Take a peek at the Assistant window—you should find a reassuring message confirming that your program is working perfectly, which is great considering we just ran it successfully!

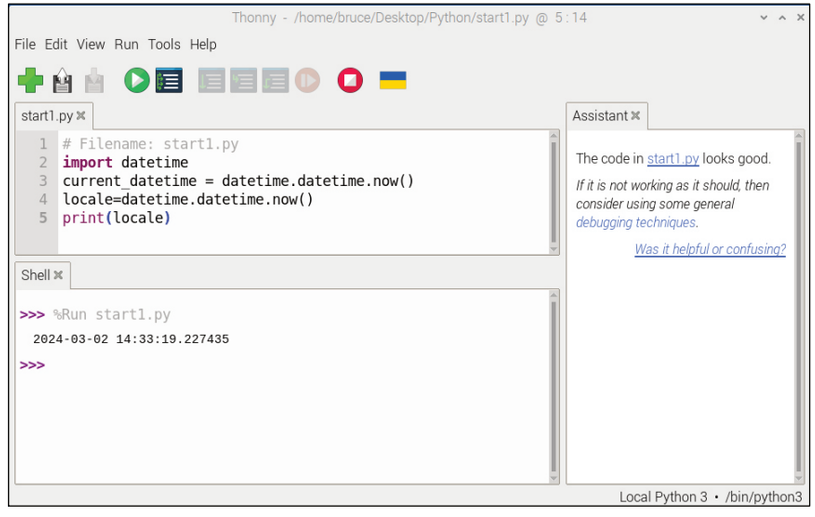


Figure 2f. Date and Time in the Shell window.

You can also run this program in the Shell window of the IDE. Follow these steps:

1. In the Code Editor, select all three lines of the program by clicking and pressing **<Ctrl-A>.** Then, press **<Ctrl-C>** to copy.
2. Navigate to the Shell window, click inside to select it, and press **<Ctrl-V>** to paste the program there.
3. Finally, press **<Enter>** to run the program.

The same process applies when using the Python Interpreter in the Terminal window. Select the Terminal window, choose 'Paste' from the 'Edit' menu, and press <Enter> to execute the program.

In both scenarios, every line you enter in either the Thonny Shell or the Python Interpreter in the Terminal window gets executed immediately. You'll see an output whenever a line involves an action that displays a result.

Things to Note: Here' are a few things to note about the program above.

- The first line starts with a **hash symbol, '#'**. A Python program ignores anything after a line that starts with a hash symbol. This allows you to put comments or notes in your programs. As you can see, I use this to denote the filename of the program. Thus, the program here is called 'start1.py'. You'll find it listed as such in the download programs. You can use as many comment lines as you wish. Don't go overboard otherwise you lose the program within the

comments. And for the scope of this book, you don't need to type them in, if you are doing that.

- The program shows how we have used an imported item, by the name of **datetime**. This item is called a **module** and contains the routines we needed to retrieve and print the date and time. A module is a file that contains Python code—which you can use in your own programs. For example, the **datetime** module helps you work with dates and times."
- The last lines gather information required and then displays it, and while you might now know the exact syntax of the rest of the program means you can read it and understand what is happening.

Docstrings

Docstrings are a way documentation for your Python programs. They can be displayed and read externally when you want to know how the program works, provided you have followed standard guidelines. Docstrings are lines of comments, started by a triple quote and a triple quote at the end, which may be many lines further down.

We'll look at how to create and use these much later in this book, so as not to clutter up the listings displayed as we progress through the book."

Thonny Windows

Thonny is flexible because it has several additional windows you can open and display. You can explore these by selecting the **View** menu at the top of the window. A good one to add is the **Files** window. You can navigate to your programs and load them with a double click, as shown in Figure 2e below. Note how the 'Files' window that the Python logo signifies Python programs.

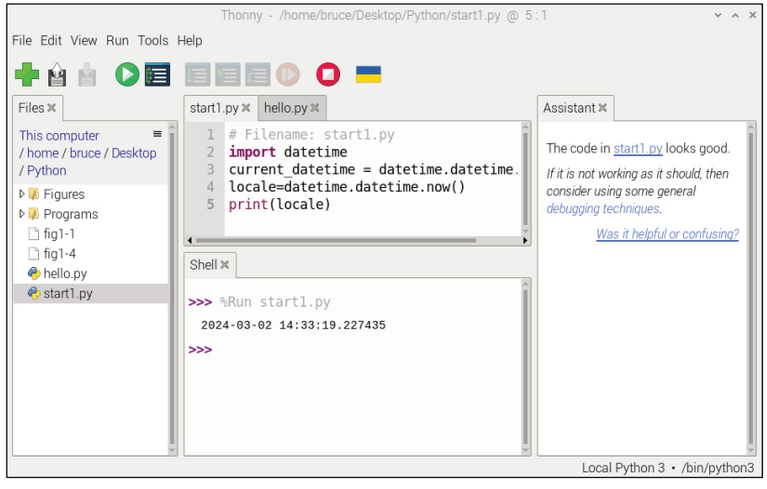


Figure 2g. Additional Thonny windows. Files is useful to keep open.

Important Line Wraps

Many of the lines of Python code in the rest of these books are too long to sit on one line. They therefore wrap into a second or more. If your type a program in and there is an error when you run the program than look at the error message and see if you can solve the issue for yourself. This may simply be deleting the <Enter> you have inserted as part of the line wrap.

Eric

Python comes with its own IDE. It's called IDEL. That's probably as much as I want to say, as that's how it should be left.

Block Indendation

Python programs will invariably contain indentations. This occurs after certain keywords and helps to structure a program and make it easier to see blocks of code. We have not encountered this commands requiring indentation as of yet, but we will come across them soon.

When a program line ends with a colon (like an if, for, while, def, or class), the indented lines that follow form a block (called a "suite" in the language reference). Python uses those leading spaces t(or tab) to decide which statements belong together, so misaligned code triggers IndentationError (or TabError if tabs and spaces are mixed). This design keeps Python readable

and encourages a clean, consistent layout: you can see the program's logic at a glance because the visual shape mirrors the control flow.

Style-wise, follow the community convention of four spaces per indentation level and avoid mixing tabs with spaces—your editor can insert spaces automatically when you press Tab. Keep nested blocks shallow where possible; deeply indented code is a sign it might be time to refactor into functions. If you split long statements across lines, use parentheses so continuation lines align naturally rather than relying on odd spacing. Sticking to these habits makes your code portable, avoids surprising errors, and helps others (and your future self) understand it quickly.

Questions

1. What command do you use to start the Python interactive shell from the Command Prompt window? And how do you exit the Python interactive shell and return to the regular prompt?
2. Which command should you run to check the version of Python on your computer?
3. What is displayed in the bottom-right corner of the Thonny window, and why is it important.

03: A Matter of Style

Okay. You might find the opening section of this chapter requires some head-scratching—especially the terminology. But give it a go, work through the chapter, and then maybe come back here a second time to understand it better. It may not seem like it now, but these concepts will become second nature without you even noticing. Many people will shudder at me putting this chapter here, and many will simply ignore it. But I just want you to know how important it is. Feel free to skip it if you are confused, but do come back to it when you have completed the first dozen or so chapters.

Object-Oriented Programming

OOP is a style of programming that uses, as its name suggests, "objects" to construct programs. It allows you to model and manage the properties and behaviour of program code as 'real-world' concepts, making them more 'lifelike.'

In your mind's eye, you can start to draw comparisons. Terms such as inheritance and encapsulation may seem complex at first, but they become clear with understanding and practice. They mean the same as they do in real life. You inherit something. You encapsulate something. Other terms, such as polymorphism and abstraction, may not be as intuitive but shouldn't impede understanding or, more importantly, you're learning of Python programming.

The Four Building Blocks of OOP in Python

- Class
- Object
- Attributes
- Methods

The House Blueprint Analogy

Think of a class as a blueprint for a house. The blueprint outlines structure, layout, and features—the number of bedrooms, restrooms, garages, and pools. These are *attributes* of the class.

From this blueprint, we can create many houses, each an *instance* of the class. Some houses may differ—one may skip the pool, another may add a game room. These changes are made either directly to attributes or via *methods*, like convert_to_entertainment_room().

Encapsulation is when data and methods are bundled together in the class so that it manages its own internal state. Abstraction is when complex actions (like converting a room) are hidden behind a simple method call.

The Constructor: **__init__()**

When creating an object, its attributes are initialised using a constructor called __init__.

```
class House:
    def __init__(self, bedrooms, restrooms, garage,
basement, pool):
        self.bedrooms = bedrooms
        self.restrooms = restrooms
        self.garage = garage
        self.basement = basement
        self.pool = pool
```

This sets up the initial values for the house.

Inheritance and Polymorphism

Inheritance lets you create new classes based on existing ones. A TownHouse might inherit from House but add new features.

Polymorphism allows a method like **describe()** to behave differently depending on the object calling it. (Note this sample is just to illustrate, it actually wont works as we shall see.

```
class TownHouse(House):
    def describe(self):
        return f"This townhouse has {self.bedrooms}
bedrooms, {self.restrooms} restrooms, and is part of a
row."
```


Instance vs Class

Each object created from a class has its own attributes.

```
house1 = House(4, 2, "double", False, True)
house2 = House(3, 1, "single", True, False)
```

house1 and house2 are separate instances, with their own attribute values.

More Examples

Entertainment Room Conversion:

```
class House:
    def convert_to_entertainment_room(self, room):
        if room in self.rooms:
            self.rooms[self.rooms.index(room)] =
'entertainment room'
            print(f"The {room} has been converted into
an entertainment room.")
```

Garage Example:

```
class House:
    def open_garage(self):
        if self.has_garage:
            print(f"Garage opens. Can hold {self.
garage_size} cars.")
        else:
            print("No garage available.")
```

Another Analogy: Shapes

A Shape class may have a method for area(). A Circle class might override it:

```
class Shape:
    def area(self):
        pass


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius


# Test it
c = Circle(5)
print("Area of circle:", c.area())
```

This example prints out the result.

Wrapping Up

You may feel this chapter is too soon. It probably is. But OOP is *everywhere* in Python. Some of this chapter will stick. The rest will click into place as you move forward. Keep returning here until one day—and that day *will* come—it all makes perfect sense.

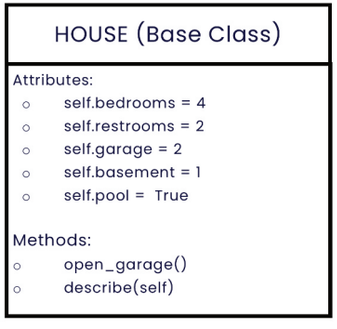


Figure 3a. The House Base Class blueprint.

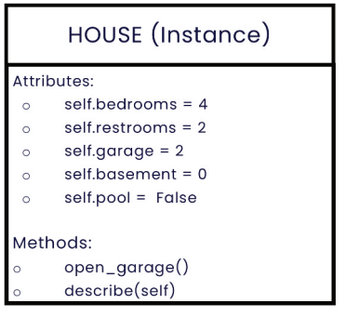


Figure 3b. Base class and an instance of a class.

Combining Snippets into a Full Program

You may feel that this chapter is way to advance to be at the front of the book. It probably is. But OOP is essential to life on Python. Some of what you have read will stick, and as I've said, and promise, these concepts will go almost unnoticed by you, and everything will fall into place as you continue. Re-read this chapter every few chapters of the book. More will stick.

These snippets of code can be combined to create a comprehensive program. While we won't combine them here (it is with the downloadable programs), I encourage you to experiment and build on these examples as you progress through the book. Keep tripping back here until you can create the completed program. Then you'll fully understand OOP!

04: Foundations

I like to think of the key components that make up a Python program as "wrappers." These wrappers are like containers for the code you write, and when you put them together in the right order, you create a fully functioning program. Imagine it like **Ferrero Rocher** chocolates—each one wrapped in shiny gold foil. In this analogy, the chocolates represent your code, and the box of chocolates is the complete program. Each wrapper (or section of your program) encapsulates a piece of code, just like those gold foils enclose a delicious treat.

In Python, these wrappers are known as **functions**, and they're created using the **def** keyword (short for define). This def keyword is essential because each function you create needs a unique name. Technically, you could have two functions with different names doing the same thing, but having multiple functions doing the same thing is not a great practice due to redundancy and maintainability issues. Plus it's just plain silly. Here's the basic syntax for defining a function:

```
def function_name(parameters):
    #Function body
    #Perform desired operations
    #Return a value if needed
```

Let's break it down:

- **def**: This keyword starts your function definition. This normally occurs inside a class definition.
- **function_name**: This is the name you give your function.
- **parameters or arguments:** These are the optional values you can pass into your function. They're enclosed in parentheses and separated by commas if there's more than one.
- **:** (colon): This signals the end of the function declaration and the start of the function's code.

- **#** (hash): This symbol allows you to place comments in the code. Everything after it on that line is ignored by Python. Note the hash is not part of the function definition. It's just a comment.
- **Function body**: This is where the magic happens—the code inside the function executes, possibly using the arguments you provided.
- **return**: If your function needs to return a value, you use this keyword. If not, you can skip it.

Using the def keyword to define functions helps you keep your code organised, making it easier to reuse sections later. Once you've defined a function, you can call it by its name followed by parentheses, which triggers the function to run and return a result if necessary. And the best part? You can call your function as many times as you like.

Here's a conceptual look at how a Python program might operate:

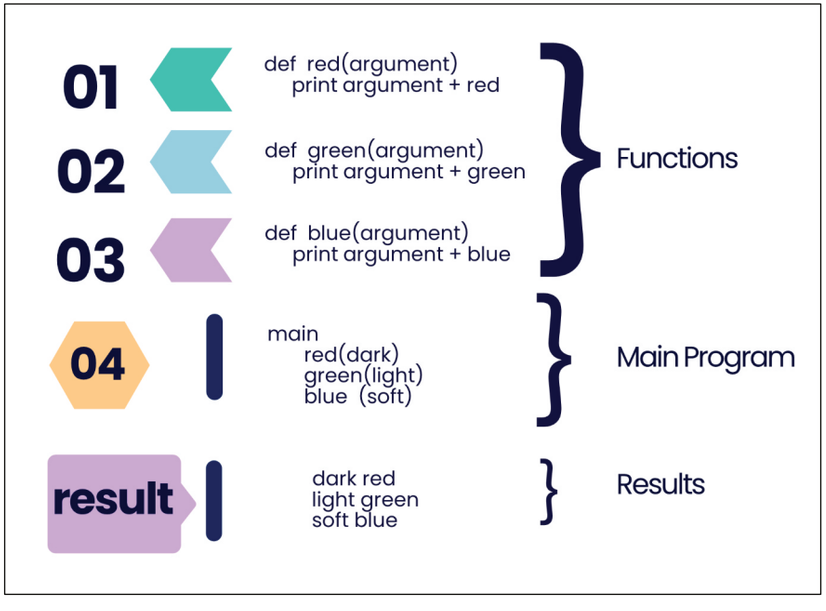


Figure 4a. Stylised version of a Python programs' operation.

In this diagram, we enter at 'main,' with each function being called and passing an argument to the next. It's not necessary to have functions right at the start of a program, but it helps keep things organised.

Each function takes its argument, performs its task (maybe even adding its own flair), and when the program finishes, it outputs the result. Hopefully, this gives you a clear sense of how a structured Python program works and

makes the whole thing easier to understand. Any Python program can be organised this way, giving your code a neat, logical flow (The numbers are the start of each line are from reference only and should not be typed in.).

```
1       #Filename: function1.py
2       def greet(name):
3             return f"Hello, {name}!"
4       message = greet("John")
5       print(message)
6       #Output: Hello, John!
```

In this example, we define a function named 'greet' with a parameter called 'name' (2). The function simply appends the value of 'name' to the word 'Hello' and then returns the combined words (3). This returned value is then stored in the container '**message**' (4), which is finally printed on the screen (5). In this specific case, the value passed as the 'name' parameter is 'John'. (Item 1 is a comment depicting the filename in the downloadable programs.)

When you run the program, it displays:

```
Hello, John
```

Let's break this down using Object-Oriented Programming (**OOP**) concepts:

Function Definition (Method):

def greet(name): defines a function called 'greet'. In OOP, functions that belong to classes are known as methods. Here, 'greet' is a method that takes a parameter called 'name'.

Function Call (Object Instantiation):

message = greet("John") is like creating an instance of an object in OOP. By calling the 'greet' method with the argument "John," you're essentially creating an instance of that function, and the result is stored in the variable container 'message'.

Printing the Output (Object Interaction):

print(message) can be thought of as interacting with an object. You're using the print command to display the result stored in 'message'. In OOP terms, this could be analogous to accessing an object's properties or calling its methods.

So, in a nutshell, this program involves defining a method ('greet'), creating an instance with specific data ("John"), and then interacting with that instance by printing the result. While this example doesn't fully dive into class-based OOP, it does touch on some core OOP principles like encapsulation (the function), instantiate (creating an instance with data), and interaction (printing the result).